CS2113 Final

AY25/26 sem 1

github.com/mendax1234

Programming Paradigms, P5

1. **(Object, P5)**: Objects in OOP is an **abstraction** mechanism because it allows us to abstract away the lower level details and work with bigger granularity entities.
2. **(Classes, P8)**:
   - **(Encapsulation vs. Abstraction)**: **Encapsulation** is the **process** of putting details and related methods together into one entity. And this entity is called the **abstraction**, as it contains less details. An object in OOP is an **encapsulation** in terms of two aspects, **P8**.
3. **(Association, P9)**: Associations between two **objects** can represent the association between two **classes**. It is usually be implemented using **instance level variables**.
   - **(Navigability, P10)**: It indicates whether one object holds a **reference** to another, allowing navigation in that direction.
     - Unidirectional: Only one class has a reference → one-way navigation.
     - Bidirectional: Both classes reference each other for the same relationship → two-way navigation; if the references represent different relationships (e.g., pet vs breeder), they are simply **two separate unidirectional** associations, not a **bidirectional** one.
   - **(Multiplicity, P10)**
   - **(Dependency, P11)**: A **dependency** is a need for one class to depend on another **without having a direct association** in the same direction. It is implemented in having **another class object** in the **method argument**.
   - **(Composition, P12)**: A strong "part-of" relationship where the **part has no meaningful independent existence outside the whole**. The implementation is to use a **`private` instance level member**.
   - **(Aggregation, P13)**: A container–contained relationship where the part is **not an integral** part of the whole and can exist independently; weaker than composition. Similar to the implementation of composition but **without `private`.**
   - **(Association class, P13)**: A special **class** that represents additional information about an **association**.
4. **(Inheritance, P14)**:
   - **(Multiple Inheritance, P14)**: A class inherited directly from multiple classes is **not allowed** in Java. Reason is that inheritance doesn't specify the override so the actual inheritance of some methods will be problematic.
   - **(Override, P15)**: Overridden methods have the same name, the same type signature, and the same **(or the subtype)** return type. a.k.a, **same method descriptor**
   - **(Overload, P15)**: Method overloading is when there are multiple methods with the **same name** but **method signatures**. Thus, these methods do the same things but take different parameters and produce possibly different return types.
   - In the code `class C {A foo (B1 x, B2 y) {}}`, `C::foo(B1, B2)` is **method signature** (method signature contains the **order** of parameters), `A C::foo(B1, B2)` is **method descriptor**.
   - **(Interface, P16)**
   - **(Abstract class, P16)**: An abstract class **cannot be instantiated**, but **can be subclassed**. A class with an `abstract` method becomes an abstract class automatically. An abstract method cannot be implemented in its own abstract class.
   - **(Substitutability, P16)**: The specification from Enzio or (LSP, **P163**)
   - **(Dynamic and Static Binding, P17)**: **Overridden methods** are resolved using **dynamic binding**, and therefore resolves to the implementation in the actual type of the object. **Overloaded methods** are resolved using **static binding**.
5. **(Polymorphism, P18)**: Polymorphism allows you to write code **targeting superclass objects**, use that code on **subclass objects**. and achieve possibly different results based on the **actual class of the object**.
6. **Java Collections Framework**: Unified architecture for representing/manipulating groups of objects. Standardizes handling to reduce effort & increase performance.
   - **Interfaces**: Abstract data types defining *what* a collection does (e.g., `List`, `Set`).
   - **Implementations**: Concrete classes defining *how* it works (e.g., `ArrayList`, `HashSet`).
   - **Algorithms**: Reusable methods for operations like searching & sorting (e.g., `Collections.sort()`).
7. **(Java Access Modifiers)**: The following table shows the access to **members** permitted by each modifier.

Modifier	Cls	Pkg	Sub	Wld
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	×
no modifier	✓	✓	×	×
`private`	✓	×	×	×

8. **(Java Packages)**: The **package** of a **type/class** should match the folder path of the **source file**. e.g., `<source-folder>/seedu/util/Formatter.java` file) is in the package `seedu.util`
9. **Important Points**:
   - **Class Level variables**: The criteria is that the variable should be same across different instances.
   - **Overriding** is **more related to polymorphism**. **Overloading** is **less related to polymorphism**.
   - **Runtime error and Compile Time Error**:
     - To see whether a code will generate compile-error or not, we **only** see the CTT of the variable and the type casting. The casted type must be the subtype of CTT of L.H.S and should be within the type hierarchy (either subtype of supertype) of the CTT of R.H.S.
     - Run-time error judgment **only** needs us to see the RTT of the variable. We **must** ignore the type casting! So, during the run-time, we can just **ignore** the type casting and see the subtype relationship between the RTT of the L.H.S and R.H.S variable.
   - Casting to an **incompatible** type can result in a `ClassCastException` at **runtime**.
   - **Association vs. Dependency**: **association** means a "**has-a**" relationship, while **dependency** means a "**uses-a**" relationship.
   - In **Composition**, whether the **whole** can exist without **part** depends the multiplicity on the **part**.
   - **Aggregation vs. Composition**: In **Aggregation**, the part can exist **without** the whole, while in **composition**, cannot.
   - **Java Primitive Type Subtype relationship**:

$$\begin{tikzcd}
\texttt{byte} \arrow[r] & \texttt{short} \arrow[r] & \texttt{int} \arrow[r] & \texttt{long} \arrow[r] & \texttt{float} \arrow[r] & \texttt{double} \\
 & \texttt{char} \arrow[ur] & & & &
\end{tikzcd}$$

   - In Java, **narrow type conversion without explicit casting** is **not allowed** and a **compilation-error** will be generated.
   - A **class method cannot access** its **instance fields** or call other of its **instance methods**. `this` keyword has no meaning in a **class method**.
   - In Java, a field/member or method with modifier `static` belongs to the **class** rather than the specific instance.
   - An **abstract class** can have **no abstract method**.
   - **All methods** declared in an **interface** are `public abstract`, so we can omit these two. **All fields/members** declared in an **interface** are `public static final` (constant), so we can omit these three.
   - `obj instanceOf Circle` will check if the **run-time type** of `obj` is a subtype of `Circle`.
   - In **Exception Handling**, if there is a block of `catch` that is **unreachable**, a **compile-error** will be generated.

Requirements, P20

1. **(Brownfield and Greenfield Project, P20)**
2. **(Non-Functional Requirements, P20)**
3. **(Gathering requiremnts, P23)**: (Brainstorming, **P23**) ; (User Surveys, **P23**) ; (Observation, **P23**) ; (Interviews, **P23**) ; (Focus Groups, **P23**) ; (Prototyping, **P23**) ; (Product Surveys, **P24**)
4. **(Specifying Requirements, P25)**: (Prose, **P25**) ; (Feature Lists, **P25**) ; (User Stories, **P25**) ; (Use cases, **P29**) ; (Glossary, **P30**) ; (Supplementary requirements, **P30**)
5. **Important Points**:
   - **User story** is **not** detailed enough to tell us exact details of the product.
   - **User story** is a **functional requirement**, not a NFR.
   - **NFRs** describe **how** the system performs (quality/constraints), whereas **Functional Requirements** describe **what** the system does (features/behaviors). NFR shouldn't be restricted to a specific user.
   - **Examples to generate NFRs**
     - **(Latency)**: Book confirmation received within **15 seconds**. (Give a specific number!)
     - **(General UI Design)**: Scanning window at least **3cm x 3cm**.
     - **(Reliability)**: Unlock signal sent successfully **99.9%** of attempts.
   - **Useless Gathering Methods**: Relying on **introspection** (guessing since developer ≠ user), surveying the **wrong demographic**, or asking **leading/technical** questions rather than observing actual user behavior.

Design, P31

1. **(Software Design, P31)**
2. **(Design Fundamentals, P32)**:
   - **(Abstraction, P32)**
   - **(Coupling, P32)**: Coupling is a measure of the degree of **dependence** between components, classes, methods, etc. High coupling is **discouraged**. X is coupled to Y if a change to Y can **potentially** require a change in X (This is usually reflected on the **method name change**). Criteria is on **P33**.
   - **(Cohesion, P33)**: Cohesion can usually be achieved by putting related codes together.
3. **(Modeling, P35)**: UML Models, P36
   - **(Class Diagram, P38)**: (Attributes and operations notation, **P39**) ; (Association, **P41**) ; (Navigability, **P45**) ; (Multiplicity, **P47**) ; (UML Notes, **P48**) ; (Inheritance, **P50**) ; (Composition, **P52**) ; (Aggregation, **P53**) ; (Dependency, **P54**) ; (Enumeration, **P55**) ; (Abstract class & method, **P56**) ; (Interface, **P57**); (Association Classes, **P58**)
   - **(Object Diagram, P59)**: (Basic notation, **P59**) ; (Association, **P60**)
   - **(Sequence Diagram, P60)**: (Sequence Diagrams Scenario, **P60**) ; (Basic Notation, **P61**) ; (Loop, **P63**) ; (Object Creation, **P63**) ; (Object Deletion, **P64**) ; (Self Invocation, **P66**) ; (Alt and Opt, **P67**) ; (Call to static methods, **P68**)
4. **(Architecture, P69)**: (N-tier architectural style, **P71**) ; (Client-server architectural style, **P71)**.
5. **(Design Patterns, P73)**: (Singleton, **P73**) ; (Facade Class, **P75**)
6. **Important Points**
   - **Compare the coupling levels**: Follow the threes steps below
     - Count the Total Dependencies: Sum up every arrow in the diagram.
     - Check "Fan-out" (a.k.a, how many arrows going out from one object): This measures how dependent a single module is.
     - "Ripple-effect": This measures how far a change travels.
   - Coupling **decreases** testability, maintainability whil **increases** the risk of regression and the value of automated regression testing.
   - **Class Diagrams** define the general static **structure** (blueprint), whereas **Object Diagrams** capture a specific runtime **state** (snapshot) with actual instances and values.
   - In **class and object diagram drawing**, the **member name** is written together with the **multiplicity**.
   - **Overridden methods** should have a **dashed line** connecting to a **note**.
   - **Navigability** can be used in both **class** and **object diagrams.**
   - **Aggregation symbol** is **not recommended** to use in

UML diagrams.
- **Enums** drawing:
  - In class diagrams, if the enum class is small, write out all the values.
  - In object diagrams, write the enum directly in the member field of that object.
- **Keywords to draw the composition arrow**: "**parts**"
- "**+**" and "**-**" only indicates **visibility** not **accessibility**!
- **Association overrides Dependency**: If a solid association line already exists between two classes, **do not draw** an additional dashed dependency line for method parameters of the same type.
- In **class diagrams**, the **method** doesn't have an association arrow, if it has, the arrow belongs to the class! (Tut8-1)
- In **object diagrams**, we only draw static **states**. So, method calls from an object will be shown as **association**! (Tut9-2-b)
- In **object diagrams**, an Association Class instance acts as a **bridge** following defined **navigability** (Source → Assoc → Target), or as a **central connector** pointing **to both** participants if navigability is undefined.
- In **sequence diagrams**, the **arrow** pointing to which **activation bar**, that method is from that **object**. Example is at **P61**.
- In **sequence diagrams**, the **activation bar** denotes the period during which the **method**/**operation** is being executed.
- In **sequence diagrams**, optional elements like the **activation bar, return arrows** can be **omitted** to reduce clutter.
- In **sequence diagram**, **alt** is used for if-else or switch statements, while **opt** is used for if without else statements.
- The software **architecture diagram** is designed by the **architect** and **cannot** contain details **private** to a component.
- **Software design patterns** are **elegant solutions** to recurring problems in software design.

Implementation, P76

1. **(IDE, P76)**
2. **(General Code Quality, P77)**
   - Avoid long methods with more than 30 LoC, **P77**
   - Avoid deep nesting, no more than 3 levels of indentation, **P77**
   - Avoid complicated expressions, especially those having many negations and nested parentheses, **P78**
   - Avoid Magic Number, the "number" here can be some magic value, like string also, **P79**
   - Make the code obvious, use explicit casting, use parentheses/braces to group, use enumerations. **P79**
   - Structure Code Logically: Group relevant code together by using newlines, **P79**
   - Do not 'Trip Up' Reader, five points in **P80**
   - Practice "Keep it Simple, Stupid", **P80**
   - Avoid premature optimizations, **P80**
   - SLAP Hard: Avoid having **multiple levels of abstraction** within a code fragment. All low-level statements are not good also. Should be all the same high-level statements. **P80**
   - Make the happy path prominent, **P81**
   - Use nouns for things(classes) and verbs for actions(methods), **P83**
   - Distinguish clearly between single-valued and multi-valued variables(ArrayList), **P83**
   - **Variable naming convention**: Use name to explain, **P83**
   - Avoid misleading names, **P84**
   - Case statement: Always use the default branch for the intended default action, not just to execute the last option, **P85**
   - Do not reuse formal parameters as local variables, **P85**
   - Avoid empty catch blocks, **P85**
   - Minimize global variables and define variables in the least possible scope, **P86**
   - Comments should explain WHAT and WHY, but **not** HOW, **P87**
   - Refactoring, **P88**
3. **(Java Coding Standard)**
   - Names representing **packages** should be in all **lower case**. e.g., `com.company.application.ui`
   - **Class**/**enum** names must be **nouns** and written in **PascalCase**. e.g., `Line, AudioSystem`
   - **Variable** names must be in **camelCase**. e.g., `line, audioSystem`
   - **Constant** names must be all **uppercase** using **underscore** to separate words (aka SCREAMING_SNAKE_CASE).
   - Names representing **methods** must be **verbs** and written in **camelCase**. e.g., `getName(), computeTotalWidth()`
   - **Underscores** may be used in test method names using the following three part format: `featureUnderTest_testScenario _expectedBehavior()`
   - All **names** should be written in English.
   - **Boolean variables**/**methods** should be named to sound like booleans. e.g., `isSet, hasLicense()`
   - **Setter methods for boolean variables** must be of the form: `void setFound(boolean isFound);`
   - **Iterator variables** can be called **i, j, k** etc.
   - Basic **indentation** should be **4 spaces** (not tabs).
   - **Line length** should be no longer than **120 chars**. Indentation for **wrapped lines** should be **8 spaces**.
   - Use **K&R style** brackets (brace in the same line with keyword, like `while`, etc)
   - In the **switch** statement, there is **no indentation** for **case** clauses.
   - **Imported classes** should always be listed explicitly. Not `import *`.
   - **Array specifiers** must be attached to the type not the variable. e.g., `int[] a = new int[20];`
   - In loops, if/else statements, always wrap with **curly braces**.
   - The **conditional** (if statements) should be put on a **separate line**.
   - **Comments** should be indented relative to their position in the code.
4. **(Documentation, P90)**: (JavaDocs, **P91**)
5. **(Error Handling, P93)**: (Exceptions, **P93**) ; (Assertions, **P94**) ; (Exceptions vs. Assertions, **P95**) ; (Logging, **P95**.
6. **(Integration, P97)**: (Build Automation, **P97**) ; (CI/CD, **P97**)
7. **(Reuse & API, P98)**
8. **Important Points**
   - **Comment Intensity** refers the **quality**/**number** of the comments, not the **indentation** of the commments.
   - **API documentation** may contain code examples.
   - An **assertion failure** indicates a **bug in the code**.
   - Use **assertions** to indicate that the **programmer** messed up; use **exceptions** to indicate that **the user** or the **environment** messed up.
   - **Private** methods of a class **are not** part of its **API**.
   - **Design Priorities**: Define **APIs**/**Interfaces** early to establish contracts and allow **parallel development**; avoid involving customers in *technical* design decisions or adhering to rigid mandates (e.g., "must, mandatory, always" are usually flags to be considered as wrong statements).

Quality Assurance, P99

1. **(Quality Assurance/Validation vs. Verification, P99)**: (Static Analysis, **P100**) ; (Formal verification, **P100**)
2. **(Testing, P101)**
   - Test case & Testability, **P101**
   - Unit Testing, **P102**
   - Stubs, **P103**: Stubs are special classes created with some hard-coded members just for unit testing.
   - Integration testing, **P104**
   - System testing, **P105**: Take the whole system and test it against the system specification.
   - Alpha and beta testing, Dogfooding, **P105**
   - Developer Testing, **P106**: The bugs should be fixed as early as possible. Remember the graph!
   - Exploratory versus scripted testing, **P106**
   - Acceptance Testing vs. System testing, **P107**
   - Regression Testing, **P108**
   - Test Automation, **P108** ; Test Coverage, **P111**
   - **(Test Case Design, P113)**
     - The E&E rule, **P113**.
     - Positive & negative test case, **P113**.
     - Black, White, Gray box, **P113**.
     - Equivalence partition, **P114**: An EP may not have adjacent values.
     - Boundary Value Analysis, **P117**: Boundary Value Analysis and Equivalence partition are two ways to form test cases.
     - All pairs strategy, **P119**: It takes the largest product of domain sizes between any two variables as the minimum number of test cases.
     - Each Valid Input at Least Once in a Positive Test Case & Test invalid inputs individually before combining them, **P120**
   - **Important Points**
     - We **don't need to** distinguish between validation and verification. Just **do both**!
     - **Developer-testing** is **more about verification** than validation. As finding bugs in code instead of finding bugs in requirements.
     - **Formal methods** can prove the **absence** of errors.
     - **Scripted Testing** means **test cases are predetermined**. They **need not** be an executable script. However, **exploratory testing** is usually **manual**.
     - **Acceptance testing** typically has **more user involvement** than **system testing**.
     - **System testing** can include testing for **non-functional qualities**.
     - **Regression testing** need not be automated but automation is highly recommended.
     - **100% path coverage** has the highest intensity of testing.
     - In **equivalence partition** drawing, start from the **specification** and then think about the **edge cases**.

Project Management, P123

1. Git staged and modified, **P124**: `git add` is doing "staging" for the files!
2. Origin and upstream, **P128**
3. Git Pull, **P128**: `git pull` is a combination of fetch and merge.
4. Git Branch, **P129**: git commits form a timeline, this timeline of commits is called a **branch**.
5. Git commit, **P130**: Git commit contains a full snapshot of the working directory.
6. Objects and refs, **P129&P130**: `master` and `HEAD` are two example references.
7. Git Tags, **P131**: Git lets you `tag` commits with names, making them easy to reference later. A tag stays fixed to a commit.
8. De-attached HEAD example, **P132**: If HEAD ref points to a commit which is not a branch ref points to, the HEAD is de-attached.
9. Good Git Commit Messages, **P134**
10. (Git Branch Example, **P136**) ; (Git merge, fast-forward, squash commit, **P137-138**) ; (Git rebasing, **P142**)
11. CS2113 Team Project Workflow, **P151**
12. **SDLC, P154**: (Sequential Models, **P154**) ; (Iterative Models, **P154**) ; (Agile Models, **P155**) ; (XP and Scrum, **P155**)
13. **(Project Planning, P158)**: (WBS, **P158**) ; (Milestones, **P159**) ; (Buffers, **P159**)
14. **Teamwork & Team Structures, P161**

Principles, P162

1. **SWE Principles, P162**): (SRP, **P162**) ; (Open-Closed Principle, **P162**) ; (LSP, **P163**) ; (SoC, **P163**)
2. **Important Points**
   - In **Singleton** design pattern, certain **classes** should have **no more than just one instance**. These s**ingle instances** are commonly known as **singletons**.
   - **LSP** states that a **subclass** should not be more restrictive than the behavior/specifications specified by the **superclass.**